RECONDO

Data Provisioning Tool

Parker Epps
Kiersten Gaspar
Brendan Pattison

6/17/2019

Table of Contents

Introduction

Recondo is a company that provides applications for hospitals and other health care facilities to manage patient care and revenue. Some examples of services that Recondo develops for healthcare providers are automated processes for checking levels of insurance coverage before and after services to prevent coverage denials, helping staff handle healthcare authorizations required by healthcare payers, replacing manual prices estimates with accurate calculations to provide price transparency, and checking the status of insurance claims. Currently, Recondo has several different applications for both client side and internal services. Due to multiple applications having been developed at various points, their corresponding databases often contain overlapping information while having to be stored in separate places. This introduces the challenge of maintaining data integrity and consistency across these databases. These products are also outdated and not in line with the Recondo's recently developed style guide and design philosophy, which focuses on more simple and modern aesthetics featuring their brand colors.

The goal of this project is to create a framework for a new tool that could be used by Recondo's clients as well as internally. It consolidates the current tools into one, while maintaining the functionality of each. It seeks to improve upon the workflow of these tools to better suit the users as well as be a more aesthetically pleasing application. The scope of this project focuses on implementing the data administration tool and the HIS Payer Mapping data type so that a simplified but stable foundation can be built. The data administration tool is currently used by Recondo employees to help healthcare facilities manage various insurance and payment data mappings.

In order to achieve these goals, both a frontend client side web-based application and a backend server side database manager was built. The frontend is built using Angular and Angular Material components to structure the necessary JavaScript functionality and HTML design elements. The backend consists of a Maven managed Java project built as a microservice leveraging various Java tools to access the database and make HTTP response to the frontend when it receives requests. The backend handles data validation, user authentication, and data models while utilizing a model-view-controller architecture pattern. This backend is designed to communicate with already existing databases stored on Recondo's servers. Both of these pieces are designed to be easily expandable so that the rest of the data types associated with the data administration tool as well as Recondo's other tools can be added to the web application by their developers. This is done by focusing on modular development and dynamic creation of certain components based upon the data instead of having them hard coded into the application.

Requirements

This application allows for adding new data to a database, single sign on for multiple tools, and user authentication. This application is expandable so that more types of data can be edited and user permissions levels can be added. It is a basic framework and starting point for continued

expansion and modification so the requirements focus on basic functionality and not implementing every possible tool and data type available.

Functional Requirements

- Application is secure - enables an auditing process by tracking which users make changes. The application therefore includes different permission levels that allow only certain people to edit certain data. It is restricted by both whom the client data belongs to and the type of data based on the user's permissions.
- Data validation occurs on both the client and server side of the tool.
- If there are multiple data stores involved, data remains consistent between databases - there are no inconsistencies or inaccuracies in data when looking at the different databases. The integrity of data is maintained.
- Data Search - user is able to select what kind of data to work with, and the data is annotated and allow for multiple search options.
- Single sign-on - users only need to sign in once and then have access to all pages and tools permitted by their clearance.
- Editing options - users are able to edit data with UI while also being fully supported by a command line interface (controls are server side - already built-in).
- Data Size - Application is able to be scaled up to satisfy the largest datasets provided by current clients (5-10 million entries) without introducing significant delays. The application is developed to ensure structuring/design can support larger data.

Nonfunctional Requirements

- Standardize interfaces - interface are consistent across users.
- Appealing Design - color scheme matches the Recondo's color palette and themes and follows good design principles for UI design.
- Modular design - different pieces are modular to allow for logical expansion.
- Efficient Access - data access and editing is efficient and does not have extensive/obtrusive delays.

System Architecture

In order to better understand all the moving parts of this project, the architecture diagram has been split up into three diagrams: the general architecture, the frontend (Angular) architecture, and the backend architecture.

Shown below in Figure 1a and Figure 1b is the general architecture. Figure 1a shows the initial flow of events that occur when the user first opens up the web application. The user is brought to a login page, where the user must input their credentials. If the user is authorized to view the application, the proper permissions for the user are within the database user interface. User permissions are determined by the realm within which the user exists (Development, Production, or Testing). If a user belongs to the development realm, they have full access to CRUD (Create, Read, Update, and Delete) operations and are able to view different data. If a user belongs to the production realm and they are a client, they are only able to view data belonging to that client and are able to edit or add data if they are an administrator. If a user belongs to the testing realm, they

are granted permissions required by the nature of the testing. The login page also has single sign on abilities, which does not require the user to enter his/her credentials if the application is closed after some amount of time.

Figure 1b shows the architecture once the user has been authenticated and has access to the database UI. The user interacts with the UI to look at information and make edits as necessary. Whenever the user first chooses the data they would like to view and edit, the Angular front end makes an HTTP call to the back end. The backend determines which part of the database to access based upon this request and returns the data to the front end. If the user filters the data after it has been fetched this is handled by the Angular Material filter functionality which will automatically display only the relevant data to a search.

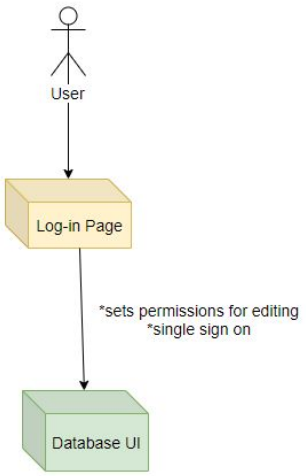

Figure 1a: Initial Architecture

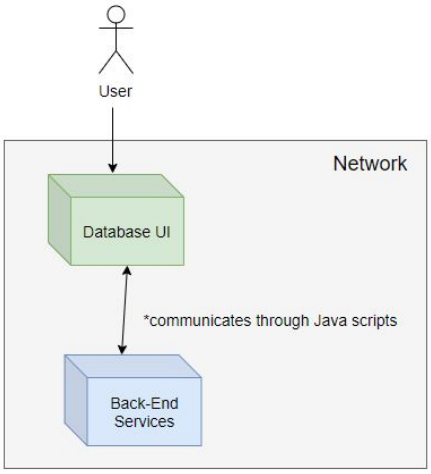

Figure 1b: Post-login Architecture

Figure 2 shows the front-end architecture of the application. The user interface is built out of Angular Material components

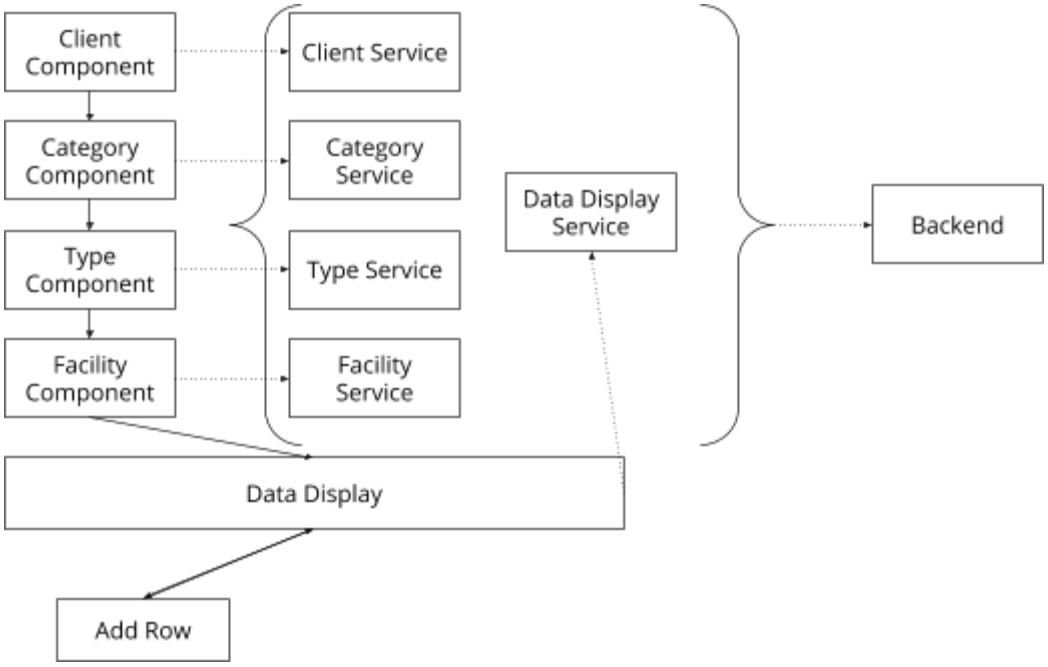

Figure 2: Front-end (Angular) Architecture

Technical Design

The backend service was implemented using Java and the IntelliJ integrated development environment. It was developed as an Apache Maven project, which allowed for managing of the project's build, reporting and documentation. It also allowed for the managing and adding of dependencies through the project object model. Swagger 2 was utilized in addition to Maven for generating the API documentation specifying the different API endpoints, the required request parameters, and the possible responses. Apache Tomcat provided a HTTP web server environment for running and testing the Java code. Subversion was used for version control of the backend source code.

It was implemented following the structure of microservice architecture, which allows for an application to exist as a collection of loosely coupled microservices, with the data provisioning tool's backend behaving as one of these microservices. This architecture pattern improves the modularity of an application, making each service easier to understand, develop, test and maintain as each service runs and is developed individually with minimal dependency on the other services. Therefore if one service fails, the rest of the services that compose an application can keep running and the entire application does not fail.

The service was implemented using the service side of a RESTful web service, with the frontend Angular application acting as the client. The client sends messages to the service via HTTP methods (ex. GET, PUT, POST, DELETE, OPTIONS), making a request to a specific uniform resource identifier (URI) and a specified method. These request methods coincide with the standard

CRUD operations for persisting data to storage. The backend then returns an HTTP response based on the URI and method of the request. In addition, the backend architecture leverages the concepts of Spring MVC frameworks, containing a model-view-controller architecture allowing for the development of a flexible application.

The model consists of plain old Java objects (POJOs) which encapsulate specified application data. The controller is responsible for processing the HTTP requests and building an appropriate model. Once a response containing data is received by the frontend from the backend, the view which is implemented by the frontend then renders the model data.

Database access was implemented using data access objects (DAOs) which provides an abstract interface to the database. Hibernate, which is an object-relational mapping framework for Java, was utilized to provide a framework for mapping the models the relational database. Hibernate also provided data query and retrieval facilities. Using these concepts and tools allowed for specific data operations without exposing the details of the database. These architecture patterns and concepts can be visualized using Figure 3 below. The figure also displays the utilities package containing other classes dealing with any methods which do not belong within a controller, model or data access object. Furthermore, each data type was organized to exist within a certain data category: HIS mapping, provider data, environment configuration, payer data, procedure data, and reference data.

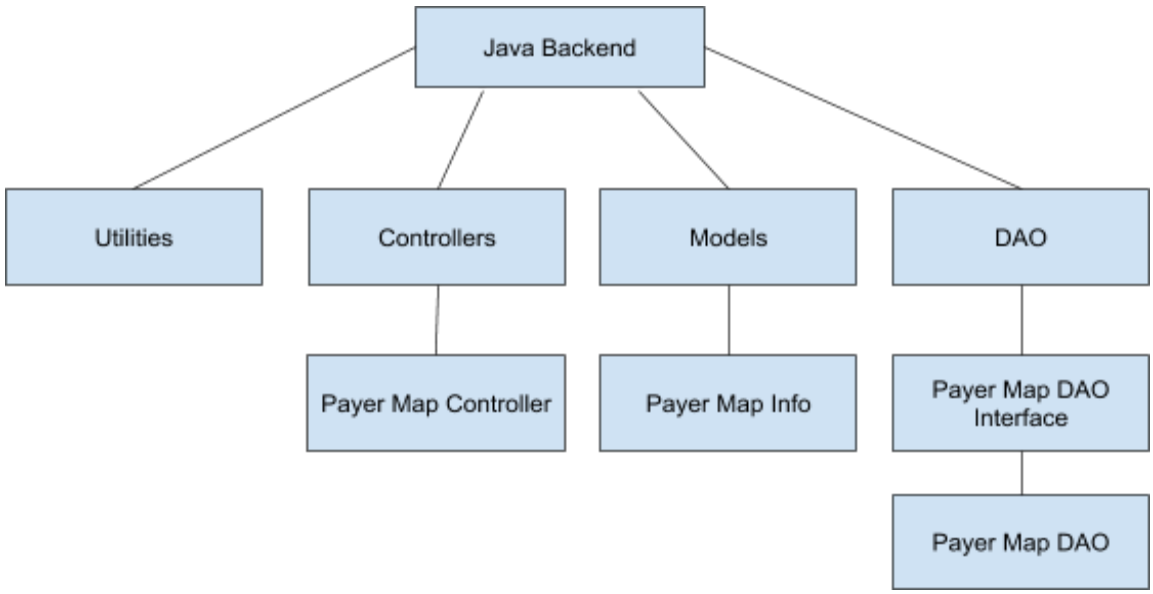

Figure 3: Backend Organization

One interesting aspect of this project was implementing dynamic field generation. Typically components like input fields in a form are hard coded, meaning the labels and types of data they receive are specified by the developer when the software is created. However, the design choice of using a new dialogue window to facilitate adding a new row of data introduced new challenges while also being good for user experience. Because different kinds of data would be displayed based upon

what was selected, the number and type of input fields required for a new data entry would vary and so would be unable to be hard coded. Therefore, the fields had to be dynamically generated based upon the currently displayed data table. Meaning different numbers and kinds of fields would be created using the information in the table so that the form matches the data.

In order to achieve this, the dialogue window utilizes the same Angular Service that the data display component does when requesting data from the backend. Due to the nature of generating the data table, two separate functions are in place for fetching the column labels and the data itself. This is useful as the dialogue window can use the same column labels for the input fields labels.

Normally each field in a form is written out individually and given its name, data type, and validation separately. In this case they need to be generated based upon the column labels given. So, an array of 'Column' objects is used that stores the display friendly name and the data label for each data column. This object is iterated over in the HTML declaration of the form fields and thus generated a new form field for each column. Passing the data input by the user is simple, because both the label and name are associated with the field and can be easily paired with the correct place in the table.

In order to add the new data to the table, the input data is passed back to data display component upon the 'Confirm' button being pressed. The data is transferred as a new data object that is the same kind as is stored in the original data file, so it can simply be pushed to the backend by the data display service using a POST command.

Another one of the interesting challenges of this project was integrating the frontend and backend services in order to have a fully functional UI hosted by a microservice backend. This functionality was not fully completed due to time constraints and scope of the project. However, the process was implemented for certain aspects of the project and can be used for extension of the project further as Recondo utilizes the framework for further development and increased functionality.

An example of this instance is the population of a facility selection dropdown displayed below in Figure 4. When the user first logs into the application, they are given a series of selection components to establish the environment and data for which the tool will be provisioning. One of these components is a facility selection component, which is populated from a call to the backend in order to retrieve a list of potential facilities to select from based on the previously selected customer. This list should not contain all facilities that Recondo carries data for, only the facilities which relate to the selected customer. The process behind the population of the list provides an example of the data flow between the frontend and backend.

Figure 4: Frontend Facility Selection Component

When a customer, data category, and data type are selected before the facility component is revealed, a frontend message service is notified of these selections and is provided the selection values. These values are passed to the frontend facilities component via the message service, which allows the facilities component to reveal itself and populate its array of facilities. The facility component in addition to the message service, includes an instance of a frontend facilities service, containing methods for making HTTP requests to the backend API through a specified URI. In this case, a GET method HTTP request is made using an imported HTTP client and an API URL ending with the term 'facilities.' The facilities service also contains a method for handling HTTP response errors which are caught through a pipe attached to the request, a useful feature of Angular. The method for retrieving the facilities expects an observable array of facilities, a simple class which models the information a facility may contain by declaring specific parameters. For example, the facility model contains a string for the facility name and a number for the facility id.

The backend when deployed receives the HTTP request made by the frontend at an endpoint specified by the URL and the method used. In this case, the Facility Target Controller Java class of the backend holds this endpoint. The endpoint exists through a method responsible for retrieving facilities with functionality added through the spring framework for correctly mapping the request and ensuring the correct parameters are included in the request. This endpoint method then passes the parameters on to a Facility Controller Java class, which extends a Base Controller class and contains several constants, validation methods, statistic gathering, and other helper methods related to facility data. The Base Controller class contains methods relevant to all controller classes, and therefore is extended by all controller classes. It also includes instances of each data access class. The Facility Controller accesses the database through a database access session and calls a method from the Facility DAO class for retrieving facility data. The Facility DAO class extends a Base DAO class containing methods relevant to all DAO classes, particularly those involving the retrieval and generation of data access sessions. The Facility DAO contains SQL statements for querying specific information and returns the obtained data to the Facility Controller.

Upon receiving the data from the Facility DAO, the controller utilizes a backend Facility Service to build data models that exists as POJOs through a set mapping scheme. An array of these objects is then returned up the call stack and sent to the frontend service as a JSON response. This information is then used to populate the facilities component so that a proper facility can be selected for data provisioning. The entire flow of this process is shown below in Figure 5.

Figure 5: Frontend to Backend Process Flow

Quality Assurance

In order to ensure a high quality product, we utilized specific testing software as well as good programming practices such as weekly client meetings, user feedback. A project tracker known as Jira is used by Recondo was also utilized to track tasks that needed to be completed as well as ensure a consistent work pace.

Protractor and Jasmine were used for end-to-end testing. Both Protractor and Jasmine are testing services that use Selenium and web drivers. These services are made specifically to test Angular components and materials, which is perfect for this project. The flow and functionality of the user interface were tested manually and automatically with Protractor. We also utilized user acceptance testing to make sure that the application is suitable for real world integration and is easy for users to understand and use well.

The three major components to our testing are verification, validation, and security. Verification involves checking that data is properly fetched, items and data fields are displayed properly, and that the tool's sign on and permissions are working correctly. Validation ensures that only proper and correct data types and formats are entered by the user. This should occur on both

the client side and server side of the provisioning tool. Therefore, the user can be informed of improper data being entered through the user interface, and the backend is protected from security vulnerabilities through SQL injection or code changes through browser tools. Security combines aspects of both verification and validation to make sure that users can only view data which they should be viewing and have permission to view. Similarly, users should only be able to edit or add new data if they have the proper authorization.

Weekly meetings with the client were used to ensure our product was still in line with their desires. As a primary component was a UI it was necessary to receive constant feedback on its design so it could iterated as the project progressed. They also ensured clarity with the client's expectations for the final product. In addition to client meetings we had the opportunity for our product to be reviewed by an employee at Recondo who uses their current data administration tool. He provided valuable insight in optimizing the workflow and confirmed that the overall design was sound and usable.

Failure to properly and adequately follow our quality assurance plan could have several ramifications. The most severe consequence of security vulnerabilities and verification of user authorization and permissions is violating the Health Insurance Portability and Accountability Act (HIPAA). Since our provisioning tool involves healthcare related data, which is both sensitive and protected by law, it is a vital concern that this information is not obtained, viewed, or changed by anyone who is not authorized. This includes disallowing healthcare providers to view healthcare information of other providers. Violations can result in fines ranging from $100 - $50,000 per violation and potential criminal charges if a violation is extreme and the result of willful neglect. Another potential ramification is an unstructured or poor framework which is difficult to maintain, since this will not be practical for use at Recondo or be extendable which is a major requirement of the tool. The tool could also be non-functioning which would mean wasted time and work for both our team and the client. Recondo will have no practical use for a non-functioning tool, particularly if the UI does not function smoothly and correctly, the backend does not properly retrieve data or persist to the database, or the two frameworks are not properly integrated.

Results

The goal of this project was to create an application which improves and standardizes Recondo's data management software by consolidating their various tools for managing patient care and revenue into one application. The application must be secure, provide valid data from both client and server side, show and create consistent data queries across all of Recondo's databases, have single sign-on abilities, and have editing options. Our final product implements all these features.

Several desired features were not implemented into this application due to the time frame of this project, and therefore were marked as 'nice to have' requirements. We did not have time to fully

implement different user permissions, which would prohibit certain individuals from viewing and/or editing certain data. Some data editing has not been implemented. Bulk editing does not exist nor does the tool that facilitates bulk editing, a command line interface.

For quality assurance testing, we used Protractor and Jasmine, a set of UI testing software that tests Angular apps through Selenium. Several tests were written to test the functionality of several features in the web application that was created. The tests cover login page functionality, table and data provisioning functionality, and the logout process. Code coverage is fairly low due to the time frame of the project. However, the tests do cover the basic functionality of the application.

If the project were to be continued outside the scope of this class, several more things would be implemented in order to make the application more functional. These would include more thorough user permission hierarchies, a command line interface for bulk editing, adding more categories of data, integrating the various tools outside of data admin, and more customization for the user about the default display options. In addition, Recondo aims to consolidate its various data tables into a more cohesive and simplified version and this would easily combine with the tool we have created.

Throughout field session we learned and practiced skills specific to working in the industry as well as skills necessary to the project. Communicating effectively with clients, making sure meetings are productive and provide useful information for both parties, as well as tracking and organizing our own work-flow were challenging to learn but vital to our success in the future.

This project required us to learn new languages that are not covered in required classes at Mines, including Javascript (specifically Angular) and HTML. We also learned how to construct a user friendly and functional web-based front end that is able to transfer data from a backend server using HTTP protocols and SQL queries to retrieve the data. Creating a cohesive front and backend was a huge learning experience as so far none of our classes provide a project like that from start to end.

While creating our frontend, we chose to use Angular Material components as they offered an easy pleasing aesthetic and simple functionality that worked with the basic HTML components. Initially, these components were easy to implement, but as we had to add more and more features to them in order to coordinate the project together, we learned how challenging it is implement multiple features of one component side by side as well as altering typically hardcoded components to be able to dynamically pull information from the backend data.

Another vital skill we learned was recognizing when and how to reduce the scope of a project. As we worked it became evident that we would not be able to implement as many data types as originally hoped due to the short time frame and lack of experience we possessed. Fortunately the team at Recondo we worked with guided us in reducing the workload to something more manageable for our team. This goes hand in hand with learning to be flexible when it comes to design choices. Our understanding of what UI design the client wanted changed several times unlike most school projects that have defined design choices made ahead of time.